PRACTICAL METHODS FOR ESTIMATING SOFTWARE SYSTEM FAULT CONTENT AND LOCATION

Allen P. Nikora
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Allen.P.Nikora@jpl.nasa.gov

Norman F. Schneidewind
Code IS/Ss
Naval Postgraduate School
Monterey, CA 93943
nschneid@nps.navy.mil

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
jmunson@cs.uidaho.edu

Extended Abstract

Over the past several years, we have developed techniques to discriminate between fault-prone software modules and those that are not, to estimate a software system's residual fault content, to identify those portions of a software system having the highest estimated number of faults, and to estimate the effects of requirements changes on software quality. The advantage of these techniques is that they can be applied during the stages of a development effort prior to test. By using these techniques, software managers have greater visibility into their projects, are able to exert more accurate and precise control over the systems for which they are responsible, and can identify and repair faults during pre-test phases at lower cost. We describe each of these techniques below.

Classification of Quality

To classify the quality of software during the quality control and prediction process, we have developed Boolean discriminant functions (BDFs) and Residual Critical Value Deviation (RCVD). Using failure data from the Space Transportation System Primary Avionics Software System (STS PASS), BDFs have been shown to provide good accuracy (i.e., 3% error) for classifying low quality software. This is true because the BDFs consist of more than just a set of metrics. They include additional information for discriminating quality: critical values. In forming BDFs, nonparametric statistical methods are used to:

1. identify a set of candidate metrics for further analysis.
2. identify the critical values of the metrics. This computation is based on the Kolmogorov-Smirnov (K-S) test.
3. find the optimal BDF of metrics and critical values based on the ability of the BDF to satisfy *both* statistical (i.e., ability to classify quality) and application (i.e., quality achieved versus the cost to achieve it) criteria.

The RCVD is based on the concept that the extent to which a metric's value deviates from its critical value, normalized by the scale of the metric, is an indicator of the degree to which the entity being measured does not conform to a specified norm. For example, the extent to which body temperature exceeds 98.6 degrees Fahrenheit is an indicator of the deviation from an established norm of human health. Measurement involves using surrogates: the deviation in temperature above 98.6 degrees is a surrogate for fever. Similarly, the RCVD is a surrogate for the extent that the quality of software deviates

from acceptable norms (e.g., zero discrepancy reports). An important aspect of software measurement is that surrogate metrics are needed to make predictions of quality early in development before quality data are available. The RCVD's application is in assessing newly developed modules by their quality in the absence of quality data. In order to initiate this process, a build of the software is used to validate the metrics to be applied to later builds. During validation, critical values are estimated by an inverse Kolmogorov-Smirnov distance criterion, as mentioned above. The validated critical values are used in subsequent builds and can be updated, if necessary, once the quality data (e.g., discrepancy reports) become available.

Structural Evolution

If a software system's structural evolutionary and failure histories during development are available, this information can be used to construct a detailed map of the system's residual fault content at any point in time. We have previously shown relationships between the measured amount of change between two successive versions of a software module and the number of faults inserted into that module, thereby providing an estimate of the rate of fault insertion. This lets us estimate the number of faults inserted into each module of the system at any point during its development. The number of residual faults in each module is computed by subtracting the number of faults known to have been repaired in a module (taken from the system's failure history) from the estimated number of faults inserted into that system. If the system's failure history is not available, a module's proportional fault burden can still be computed using its measured structural evolution. In this case, a module's fault burden will be proportional to the total amount of change it has received, divided by the total amount of change the system has received. Software managers can use this information to more accurately prioritize those modules to which fault identification and repair resources should be applied, thereby making the most effective use of their resources.

Requirements Risk

One of the problems during software maintenance is to evaluate the risk of implementing requirements changes. These changes can affect the reliability and maintainability of the software. To assess the risk of changes for the NASA Space Transportation System flight software, the software development contractor uses risk factors, including:

- Number of times the change was presented to the Change Control Board before being approved
- Whether the change was on a nominal or off-nominal path
- Whether the change affects an area of the software critical to mission success
- Number and types of other requirements affected by the given requirement change

The risk factors were identified by agreement between NASA and the development contractor based on assumptions about the risk involved in making changes to the software. To date this qualitative risk assessment has proven useful for identifying possible risky requirements changes or, conversely, providing assurance that there are no unacceptable

risks in making a change. However, there has been no quantitative evaluation to determine whether, for example, high risk factor software was really less reliable and maintainable than low risk factor software. In addition, there is no model for predicting the reliability and maintainability of the software, if the change is implemented. We are working both of these issues. We had considered using requirements attributes like completeness, consistency, correctness, etc as risk factors. While these are useful generic concepts, they are difficult to quantify. Although some of the risk factors also have qualitative values assigned, there are a number of quantitative factors, and many of the factors deal with the execution behavior of the software (i.e., reliability), which is our research interest.

The Need for Tools

There are practical issues that must be addressed prior to implementing these methods on a software development effort:

1. Because of the volume of data involved, tools must be used to take the measurements needed to form BDFs and RCVDs, or to measure the history of the system's structural evolution. Although there are many tools for measuring source code during implementation, the measurements taken by these tools are not standardized. For instance, each tool may have a different definition of what constitutes an operator, and of what constitutes an operand. We have developed a standard for measuring C and C++ source code; the tool we are currently inserting into development efforts at the Jet Propulsion Laboratory (JPL) takes measurements according to this standard. When using this tool, we will always know how the measurements were taken, and will better to be able to more precisely determine a particular structural characteristic's relationship to fault content.

   In measuring a system's structural evolution, it is also necessary to have a measurement process that is minimally intrusive. We have found that asking developers to perform additional activities to measure their workproducts results in incomplete, inconsistent, and inaccurate measurements. As a solution to this problem, we have developed a set of scripts and a metrics repository that integrate with the configuration management tool used by several development efforts at JPL. Together with the configuration management policies that are being defined, the scripts will invoke the measurement tools in a manner that is transparent to the developers and append the measurements to the repository. The repository itself is defined and implemented in a manner that will make it easy for developers to view the measurements and relate those measurements to the quality of their workproducts.
2. We have found it significantly more difficult to measure artifacts produced in earlier development phases than to measure source code. In many development efforts, we have observed that the syntax of the notations used in producing designs and specifications is not as well defined as that of the source code, making it difficult to define a complete or consistent set of measurements. In many cases, designs and specifications are specified in a mixture of natural language and other informal or semi-formal notations. This compounds the problem by introducing the possibility of incompatibilities between the notations. To resolve this issue, we are currently investigating

methods of translating the outputs of some of the more popular tools for diagrammatically representing a system's behavior (i.e., statecharts) into forms that can easily be measured.

3. In estimating the rate of fault insertion, it is necessary to trace repaired faults back to their insertion point, so that a proper estimate of the fault insertion rate can be obtained. However, failure histories often do not directly identify the faults that were repaired – information is limited to a description of the erroneous behavior and identification of the modules(s) that were repaired. Calibration of the fault insertion model requires that the fault data be at the same level of granularity as the structural information, i.e., at the level of individual modules. Since failures can span multiple modules, the number of observed failures cannot be used as a fault count surrogate – the underlying faults themselves must be identified and counted. Identification of faults within a module may proceed from examining the differences between successive revisions of a module, provided that the faults have been repaired in the later revision, but not in the earlier one. This requires a taxonomy allowing us to identify faults unambiguously and repeatably within these differences. During our previous work, we defined a taxonomy based on the types of changes made to a module in response to reported failures. We are currently refining this taxonomy and determining how it might be formalized.

Conclusion

The above methods can make use of software measurements available prior to implementation, thereby allowing faulty modules to be identified during early development phases. This is especially appealing since it has been repeatedly demonstrated that removing faults during the latter phases of a software development effort can be one or two orders of magnitudes more costly than removing those same faults during earlier development phases.

Acknowledgements

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology. Portions of the work were sponsored by the National Aeronautics and Space Administration's IV&V Facility